UITableView
UITableViewController

UITableView

UITableView– lists which allow user to select, delete or reorder list items

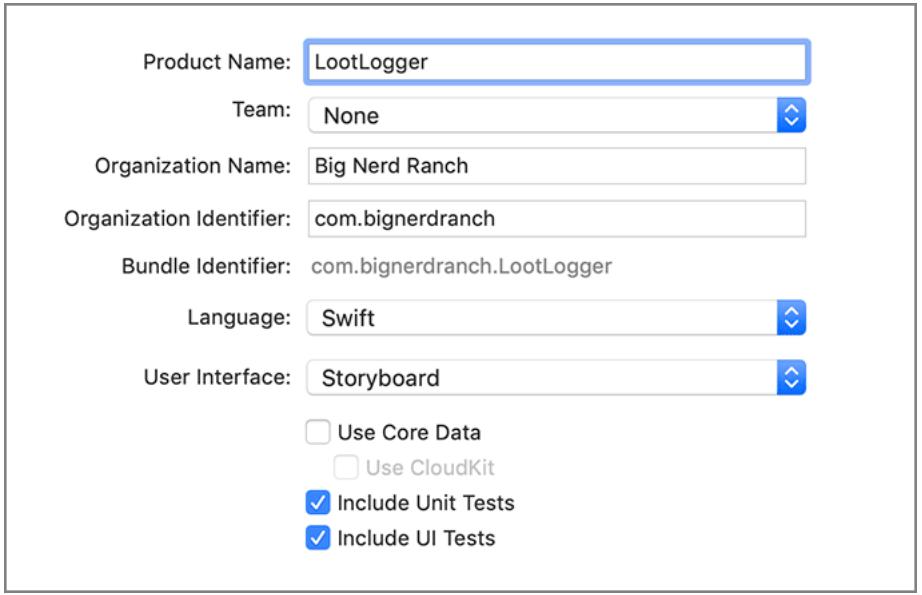
- address book
- Contacts
- Albums

a column with a number of rows

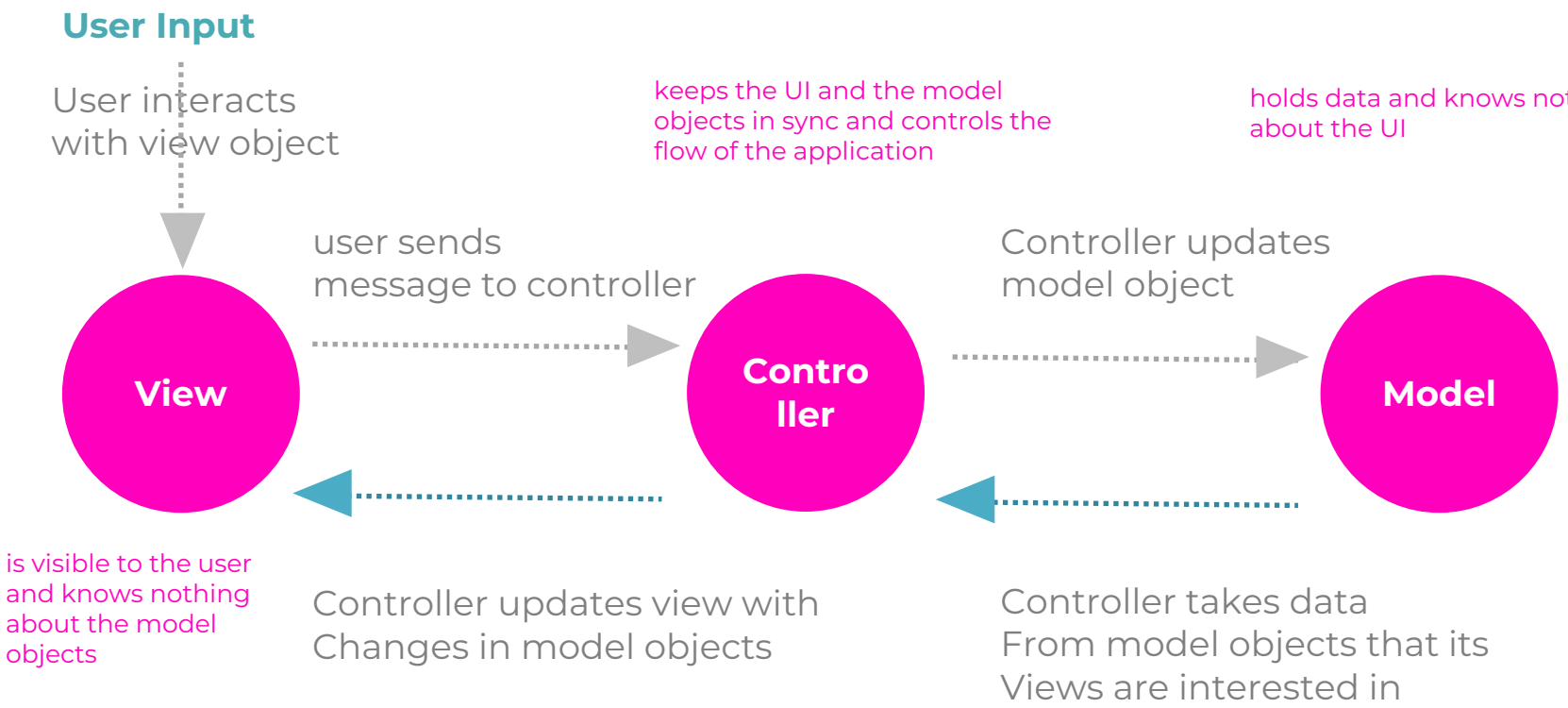
LootLogger App

Inventory in case of emergency for your insurance company

UITableView



MVC – model view controller



UITableView

A UITableView is a view object.

- **needs a view controller to handle its appearance on the screen.**

- **needs a data source.**
  A UITableView asks its data source for the number of rows to display, the data to be shown in those rows, and other tidbits that make a UITableView a useful UI. Without a data source, a table view is just an empty container. The dataSource for a UITableView can be any type of object as long as it conforms to the UITableViewDataSource protocol.

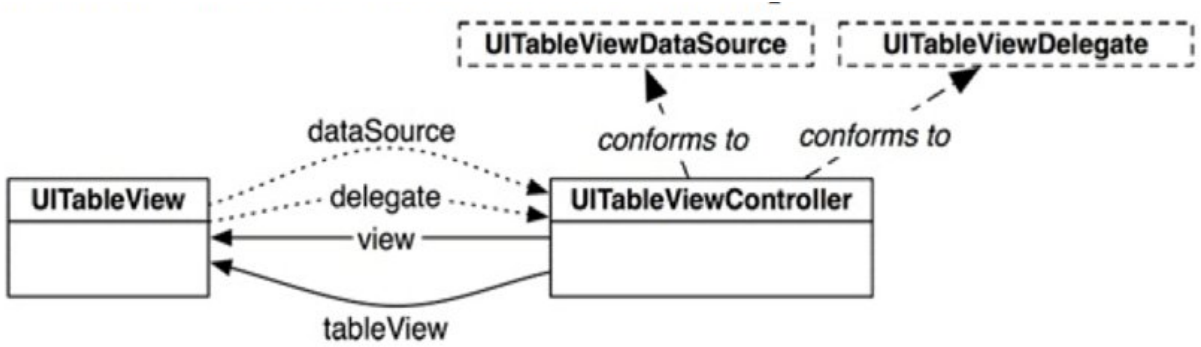
- **needs a delegate that can inform other objects of events involving the UITableView.**
  The delegate can be any object as long as it conforms to the UITableViewDelegate protocol.

UITableView

UITableViewController is a subclass of UIViewController and therefore has a view.

An instance of the class UITableViewController can fill all three roles:
view controller,
data source,
and delegate.

A UITableViewController's view is always an instance of UITableView,
UITableViewController handles the preparation and presentation of the UITableView.
When a UITableViewController creates its view, the dataSource and delegate properties of the UITableView are automatically set to point at the UITableViewController.



UITableView

Createa new Swift file named ItemsViewController and implement a subclass of UITableViewController

In ItemsViewController.swift, define a UITableViewController subclass named ItemsViewController.

```
import UIKit

class ItemsViewController: UITableViewController {

}
```

UITableView

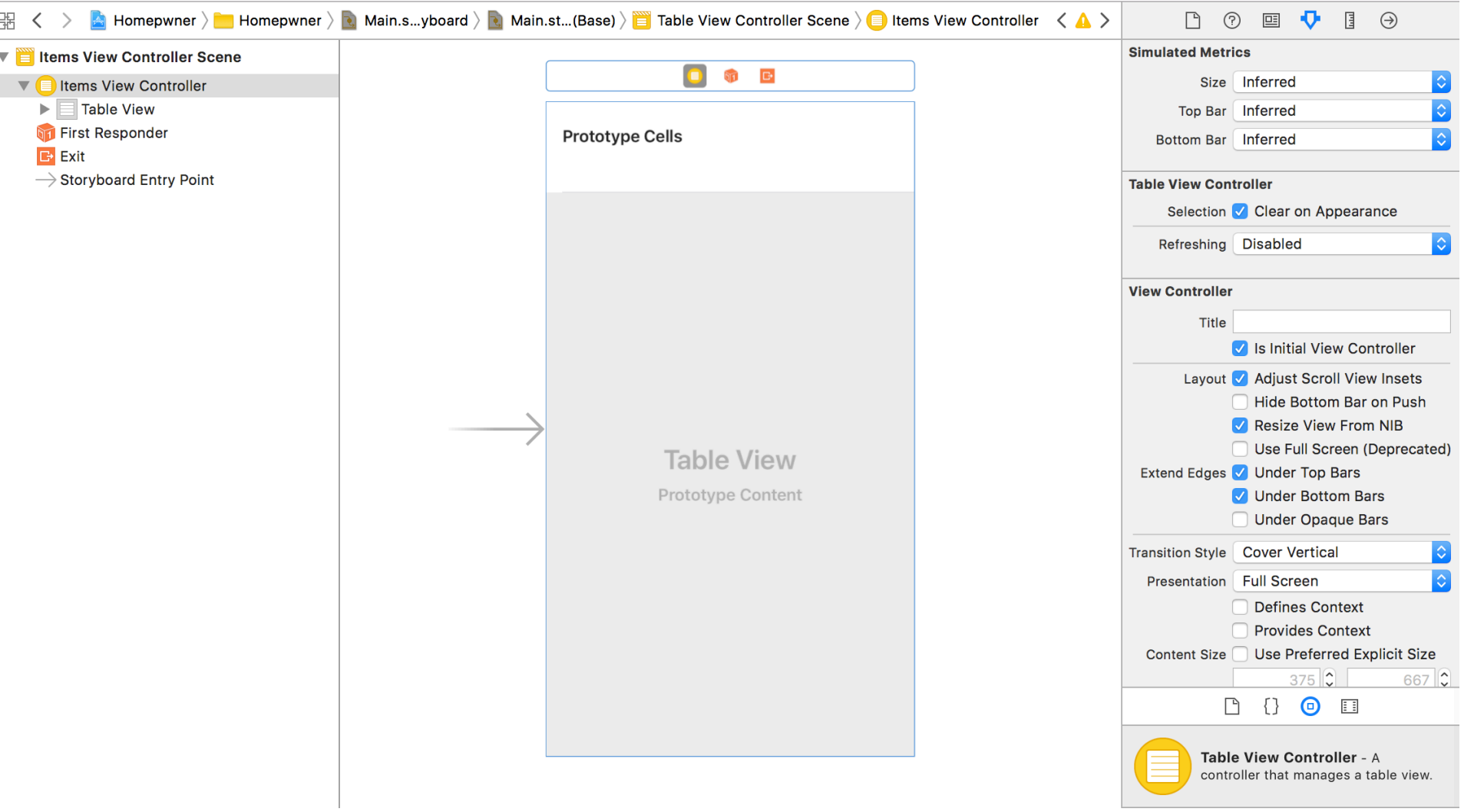
open Main.storyboard and set a table view controller.

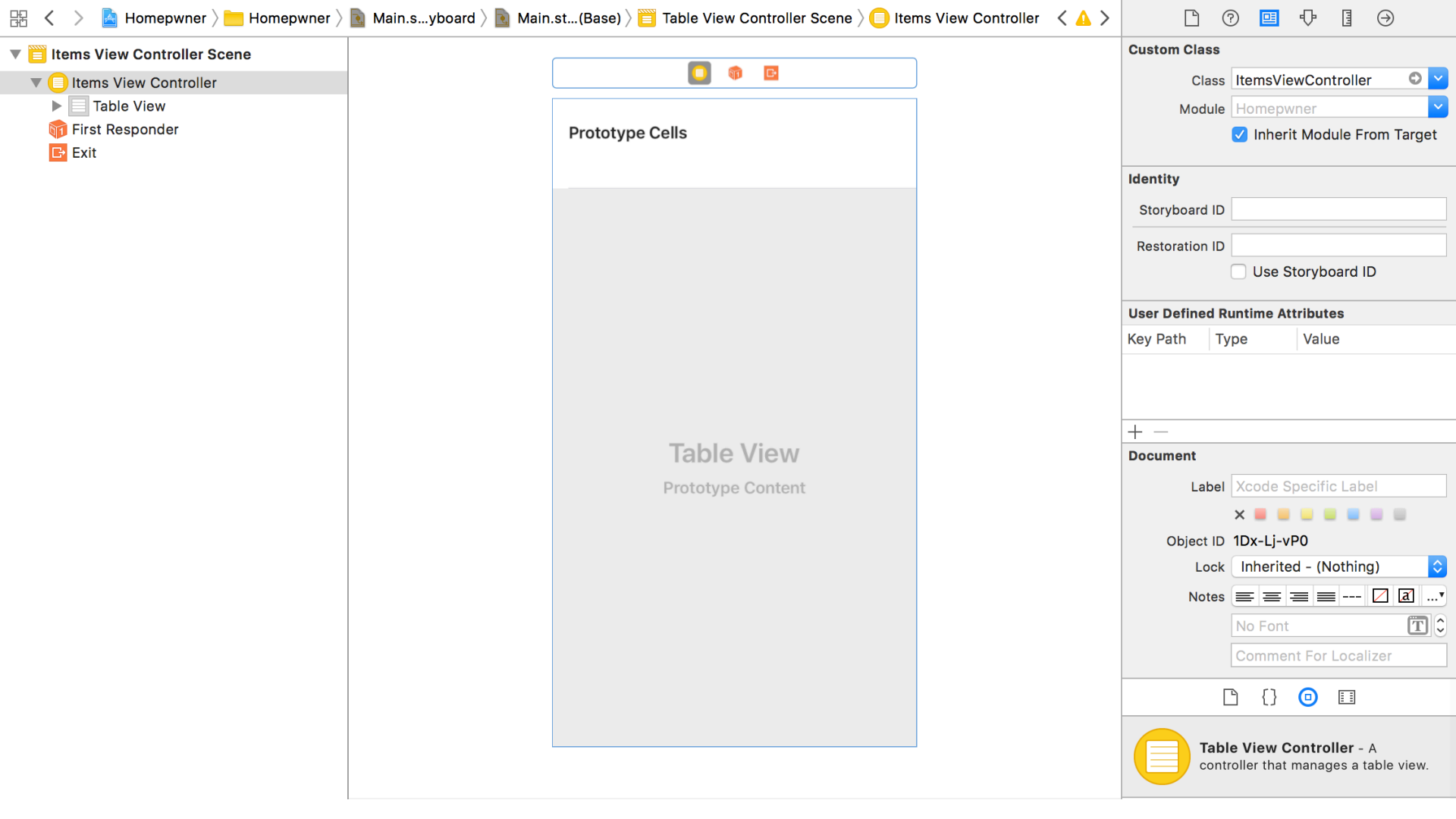
Select the existing View Controller on the canvas and press Delete.

drag a Table View Controller from the object library onto the canvas.

With the Table View Controller selected, open its identity inspector and change the class to ItemsViewController.

open the attributes inspector for Items View Controller and check the box for Is Initial View Controller.

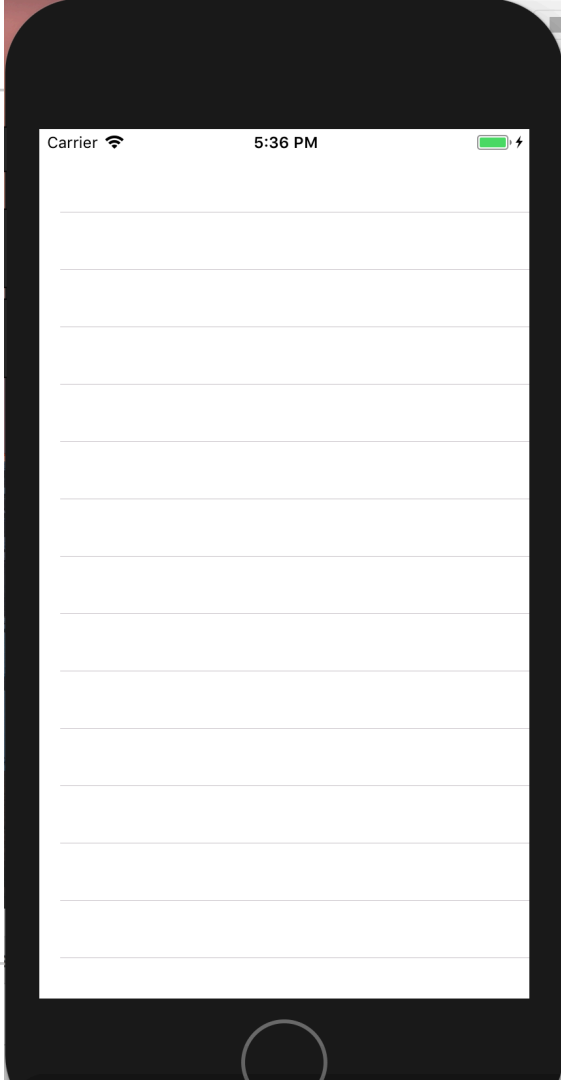

Homepwner
Homepwner
Main.s...yboard
Main.st...(Base)
Table View Controller Scene
Items View Controller
Items View Controller Scene
Items View Controller
Table View
First Responder
Exit
Storyboard Entry Point
Prototype Cells
Table View
Prototype Content
Simulated Metrics
Size
Inferred
Top Bar
Inferred
Bottom Bar
Inferred
Table View Controller
Selection
Clear on Appearance
Refreshing
Disabled
View Controller
Title
Is Initial View Controller
Layout
Adjust Scroll View Insets
Hide Bottom Bar on Push
Resize View From NIB
Use Full Screen (Deprecated)
Extend Edges
Under Top Bars
Under Bottom Bars
Under Opaque Bars
Transition Style
Cover Vertical
Presentation
Full Screen
Defines Context
Provides Context
Content Size
Use Preferred Explicit Size
375
667
Table View Controller - A
controller that manages a table view.


Homepwner
Homepwner
Main.s...yboard
Main.st...(Base)
Table View Controller Scene
Items View Controller
Items View Controller Scene
Items View Controller
Table View
First Responder
Exit
Prototype Cells
Table View
Prototype Content
Custom Class
Class
ItemsViewController
Module
Homepwner
Inherit Module From Target
Identity
Storyboard ID
Restoration ID
Use Storyboard ID
User Defined Runtime Attributes
Key Path
Type
Value
Document
Label
Xcode Specific Label
Object ID
1Dx-Lj-vP0
Lock
Inherited - (Nothing)
Notes
No Font
Comment For Localizer
Table View Controller - A
controller that manages a table view.

UITableView

Build and run your application to see an empty table view.

Delete ViewController.swift file



Creating the Item Class

Each row in the table view will display an item with information such as a name, serial number, and value in dollars.

Create a new Swift file named Item. In Item.swift, define the Item class and give it four properties. Xcode will complain that there is no initializer. Which we will add in the next step.

```
import UIKit


class Item {
    var name: String
    var valueInDollars: Int
    var serialNumber: String?
    let dateCreated: Date
}
```

Creating the Item Class

Classes can have two kinds of initializers:
designated initializers and convenience initializers.

A designated initializer is a primary initializer for the class.
Every class has at least one designated initializer.
A designated initializer ensures that all properties in the class have a value.
Once it ensures that, a designated initializer calls a designated initializer on its superclass (if it has one).

Implement a new designated initializer on the Item class that sets the initial values for all of the properties.

Creating the Item Class

```
import UIKit
class Item {
    “var name: String
    var valueInDollars: Int
    var serialNumber: String?
    let dateCreated: Date

    init(name: String, serialNumber: String?, valueInDollars: Int) {
        self.name = name
        self.valueInDollars = valueInDollars
        self.serialNumber = serialNumber
        self.dateCreated = Date()
    }
}
```

This initializer takes in arguments for the name, serialNumber, and valueInDollars. Because the argument names and the property names are the same, you must use self to distinguish the property from the argument.

Creating the Item Class

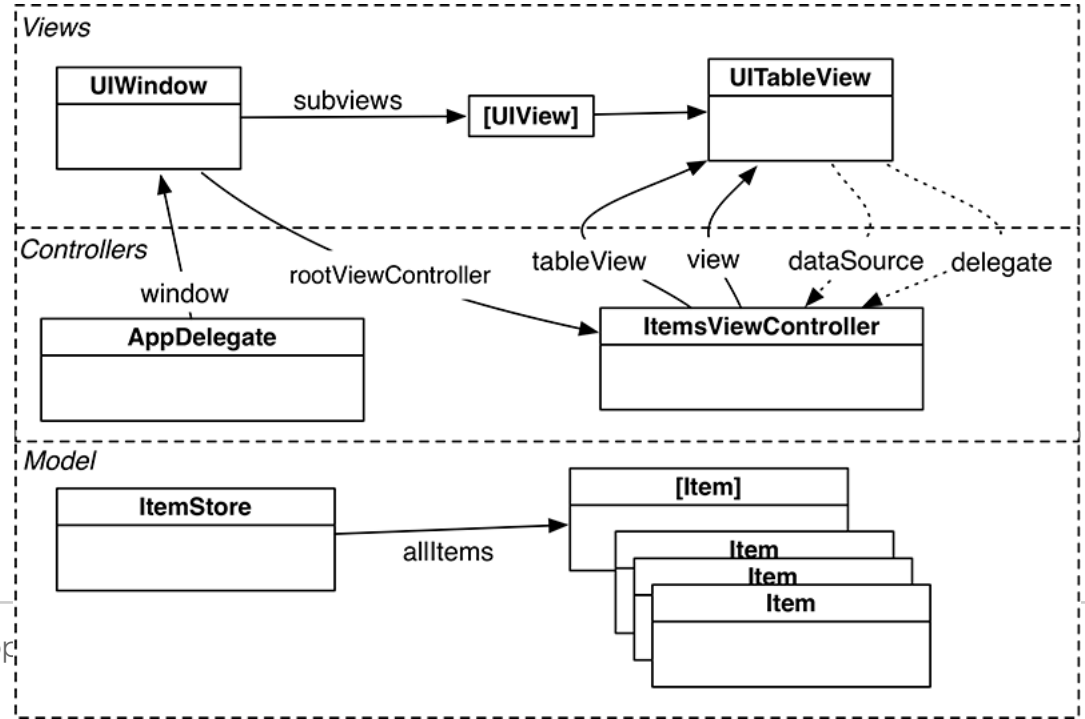
Add convenience initializer - a helper.
A convenience initializer always calls another initializer on the same class.
Convenience initializers are indicated by the convenience keyword before the initializer name.
Add a convenience initializer to Item that creates a randomly generated item.

Creating the Item Class

```
convenience init(random: Bool = false) {
    if random {
        let adjectives = ["Fluffy", "Rusty", "Shiny"]
        let nouns = ["Bear", "Spork", "Mac"]

        let randomAdjective = adjectives.randomElement()!
        let randomNoun = nouns.randomElement()!

        let randomName = "\(randomAdjective) \(randomNoun)"
        let randomValue = Int.random(in: 0..<100)
        let randomSerialNumber =
            UUID().uuidString.components(separatedBy: "-").first!

        self.init(name: randomName,
                  serialNumber: randomSerialNumber,

valueInDollars: randomValue)
    } else {
        self.init(name: "", serialNumber: nil, valueInDollars: 0)
    }
}
```

UITableView's Data Source

To provide rows to a UITableView in Xcode, the table view asks another object – its dataSource – what it should display.
In this case, the ItemsViewController is the data source, so it needs a way to store item data. This example uses an array to store the Item instances, but with a twist. The array that holds the Item instances will be abstracted into another object – an ItemStore.



UITableView's Data Source

If an object wants to see all of the items, it will ask the ItemStore for the array that contains them. Later the store will be responsible for performing operations on the array, like reordering, adding, and removing items., also for saving and loading the items from disk.

Create a new Swift file named ItemStore.

In ItemStore.swift, define the ItemStore class and declare a property to store the list of Items.

ItemStore is a Swift base class – it does not inherit from any other class. Unlike the Item class that you defined earlier, ItemStore does not require any of the behavior that NSObject affords.

```
import UIKit
class ItemStore {

var allItems = [Item]()
}
```

UITableView's Data Source

“@discardableResult func createItem() -> Item {
    let newItem = Item(random: true)

    allItems.append(newItem)

    return newItem

In ItemsViewController.swift, add a property for an ItemStore.

”

“class ItemsViewController: UITableViewController {

    var itemStore: ItemStore!
}

UITableView's Data Source

"The SceneDelegate is declared in SceneDelegate.swift and serves as the delegate for the application's scenes.

"Open SceneDelegate.swift and locate its scene(_:willConnectTo:options:) method. Access the ItemsViewController (which will be the rootViewController of the window) and set its itemStore property to be a new instance of ItemStore.

```
"func scene(_ scene: UIScene,
        willConnectTo session: UISceneSession,
        options connectionOptions: UIScene.ConnectionOptions) {
    guard let _ = (scene as? UIWindowScene) else { return }

// Create an ItemStore
    let itemStore = ItemStore()

    // Access the ItemsViewController and set its item store
    let itemsController = window!.rootViewController as! ItemsViewController
    itemsController.itemStore = itemStore
}
```

UITableView's Data Source

Copy and paste the contents of the following files into your project:

ItemsViewController.swift

ItemStore.swift

Item.swift

SceneDelegate.swift

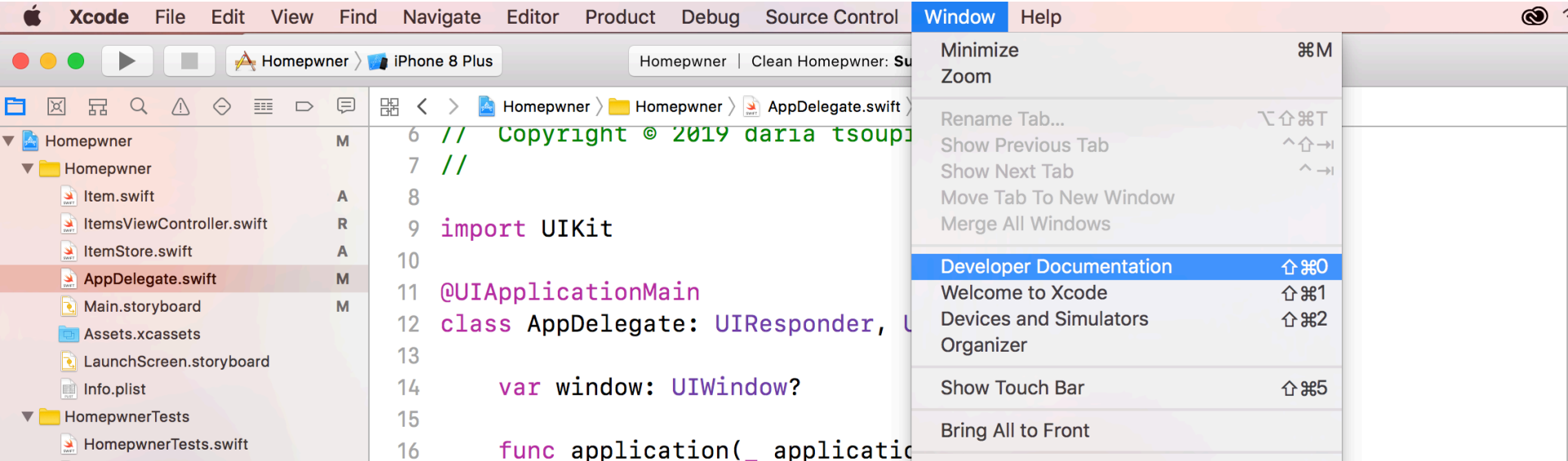
UITableView's Data Source methods

Now that there are some items in the store, you need to teach ItemsViewController how to turn those items into rows that its UITableView can display.

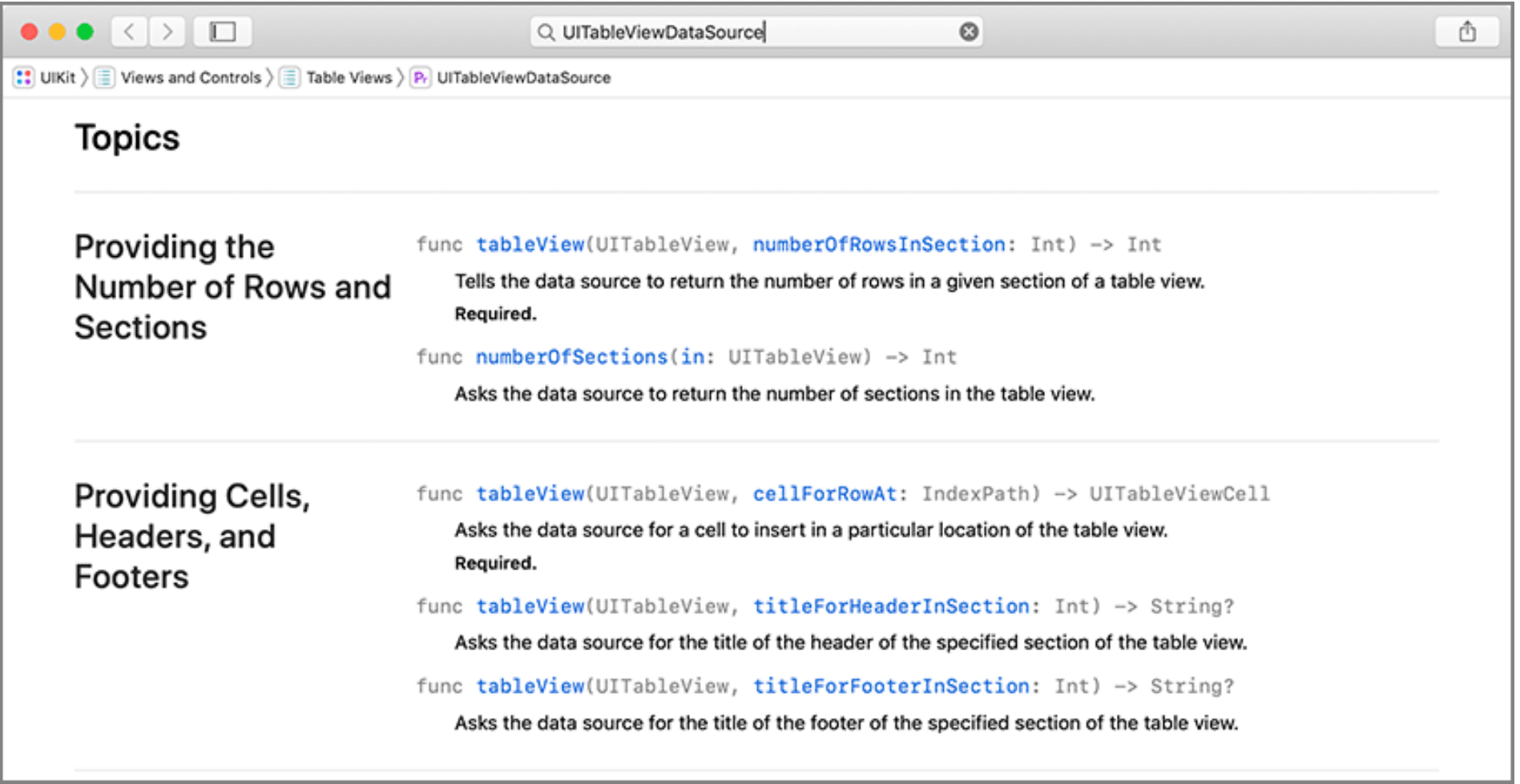
When a UITableView wants to know what to display, it calls methods from the set of methods declared in the UITableViewDataSource protocol.

Open the documentation (Window>Developer Documentation) and search for the UITableViewDataSource

Scroll down to the section titled Configuring a Table View



UITableView's Data Source methods

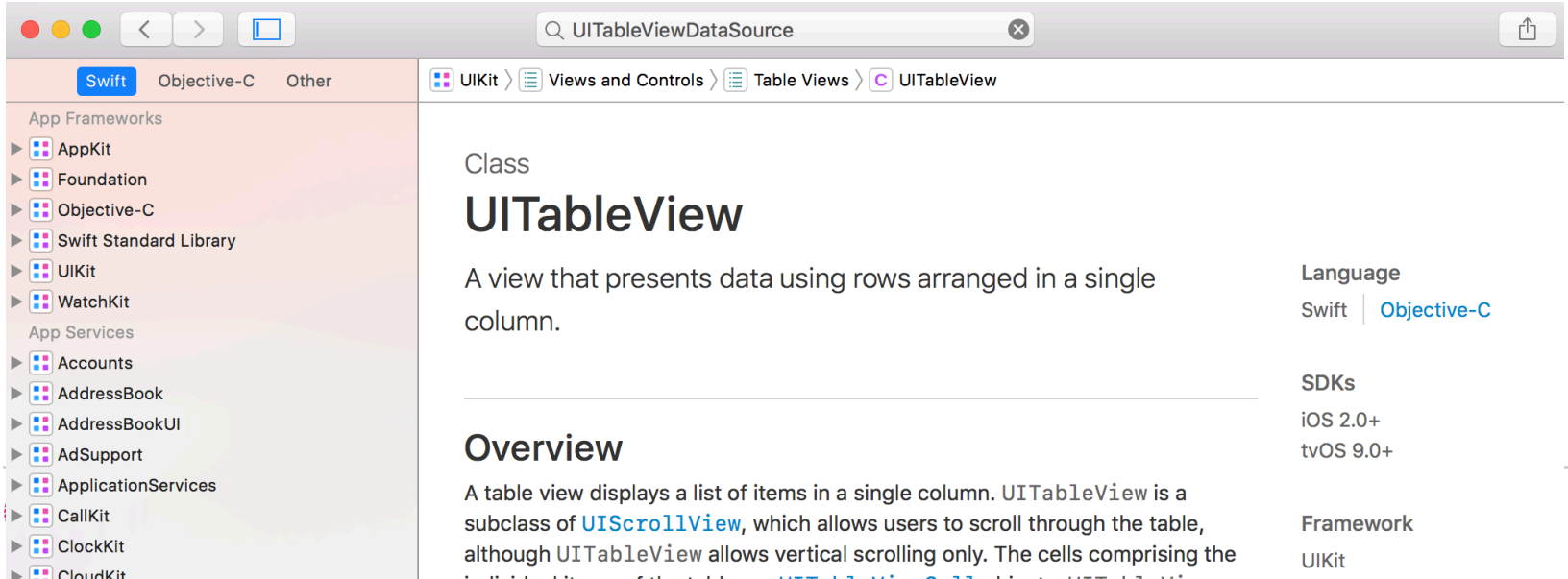


UITableView's Data Source methods

Now that there are some items in the store, you need to teach ItemsViewController how to turn those items into rows that its UITableView can display.

When a UITableView wants to know what to display, it calls methods from the set of methods declared in the UITableViewDataSource protocol.

Open the documentation (Window>Developer Documentation) and search for the UITableViewDataSource
Scroll down to the section titled UITableView



UITableView's Data Source methods

Whenever a UITableView needs to display itself, it calls a series of methods (the required methods plus any optional ones that have been implemented) on its dataSource.

The required method tableView(_:numberOfRowsInSection:) returns an integer value for the number of rows that the UITableView should display.

In the table view for Homepwner, there should be a row for each entry in the store.

In ItemsViewController.swift, implement tableView(_:numberOfRowsInSection:).

```
override func tableView(_ tableView: UITableView,

numberOfRowsInSection section: Int) -> Int {
return itemStore.allItems.count
}
```

UITableView's Data Source methods

Table views can be broken up into sections, with each section having its own set of rows.
For example, in the address book, all names beginning with "C" are grouped together in a section.

By default, the table view has one section.

But one can use multiple sections.

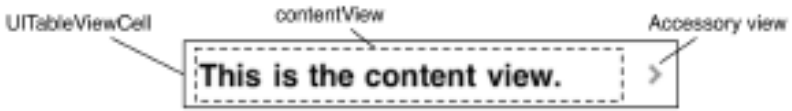
UITableView's Data Source methods

Each row of a table view is a view.

These views are instances of UITableViewCell.

A cell itself has one subview – its contentView.

The contentView is the superview for the content of the cell.



The cell may also have an accessory view. The accessory view shows an action-oriented icon, such as a checkmark, a disclosure icon, or an information button.

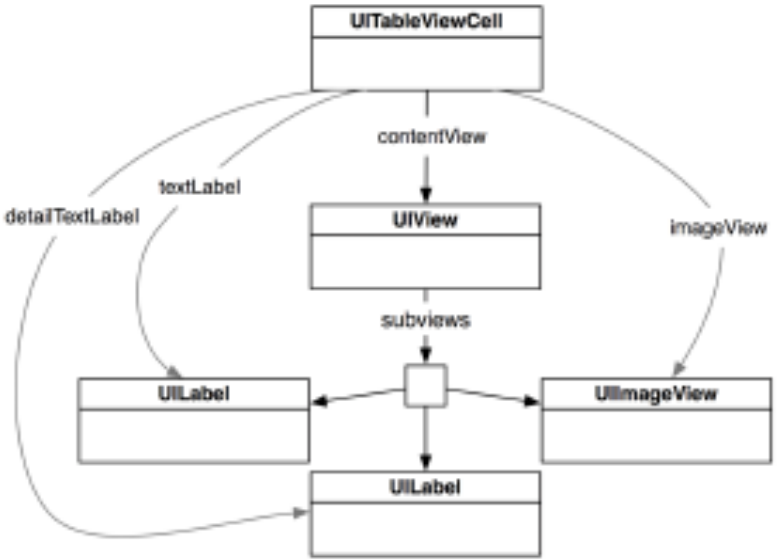
These icons are accessed through predefined constants for the appearance of the accessory view. The default is UITableViewCellAccessoryType.none.

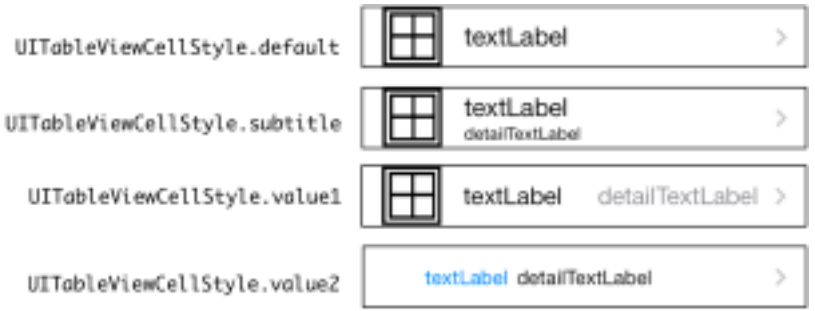
UITableView's Data Source methods

The real meat of a UITableViewCell is the contentView, which has three subviews of its own.

Two of those subviews are UILabel instances that are properties of UITableViewCell named textLabel and detailTextLabel.

The third subview is a UIImageView called imageView. Each cell also has a UITableViewCellStyle that determines which subviews are used and their position within the contentView.

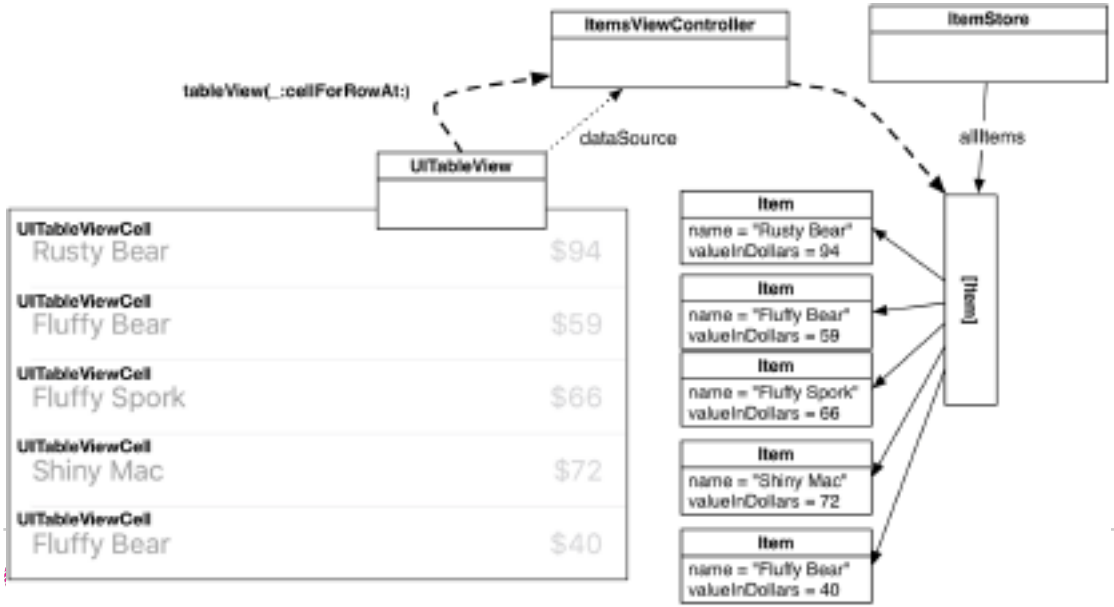




UITableView's Data Source methods

To make each cell display the name of an Item as its textLabel and the valueInDollars ofthe Item as its detailTextLabel, we need to implement the second required method from the UITableViewDataSource protocol, tableView(_:cellForRowAt:).

This method will create a cell, set its textLabel to the name of the Item, set its detailTextLabel to the valueInDollars of the Item, and return it to the UITableView.



UITableView's Data Source methods

How do you decide which cell an Item corresponds to?

One of the parameters sent to tableView(_:cellForRowAt:) is an IndexPath, which has two properties: section and row.

When this method is called on a data source, the table view is asking, "Can I have a cell to display in section X, row Y?"

UITableView's Data Source methods

In ItemsViewController.swift, implement tableView(_:cellForRowAt:) so that the nth row displays the nth entry in the allItems array.

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    // Create an instance of UITableViewCell, with default appearance
    let cell = tableView.dequeueReusableCell(withIdentifier: "UITableViewCell", for: indexPath)

    // Set the text on the cell with the description of the item
    // that is at the nth index of items, where n = row this cell
    // will appear in on the tableview
    let item = itemStore.allItems[indexPath.row]

    cell.textLabel?.text = item.name
    cell.detailTextLabel?.text = "$\(item.valueInDollars)"

    return cell
}
```

Reusing UITableViewCells

iOS devices have a limited amount of memory.
If you were displaying a list with thousands of entries in a UITableView, you would have thousands of instances of UITableViewCell.

Most of these cells would take up memory needlessly. After all, if the user cannot see a cell onscreen, then there is no reason for that cell to have a claim on memory.

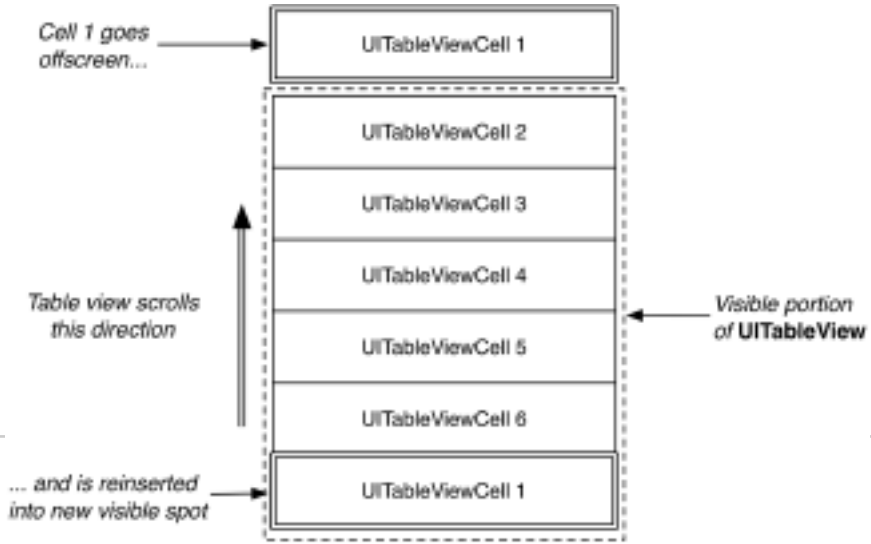
To conserve memory and improve performance, you can reuse table view cells.

When the user scrolls the table, some cells move offscreen.

Offscreen cells are put into a pool of cells available for reuse.

Then, instead of creating a brand new cell for every request, the data source first checks the pool.

If there is an unused cell, the data source configures it with new data and returns it to the table view.



Configuring the UI

Reusing UITableViewCells

To reuse cells, you need to register either a prototype cell or a class with the table view for a specific reuse identifier.

register the default UITableViewCell class to tell the table view, “Hey, any time I ask for a cell with this reuse identifier, give me back a cell that is this specific class.”

The table view will either give you a cell from the reuse pool or instantiate a new cell if there are no cells of that type in the reuse pool.

Open Main.storyboard. Notice in the table view that there is a section for Prototype Cells.

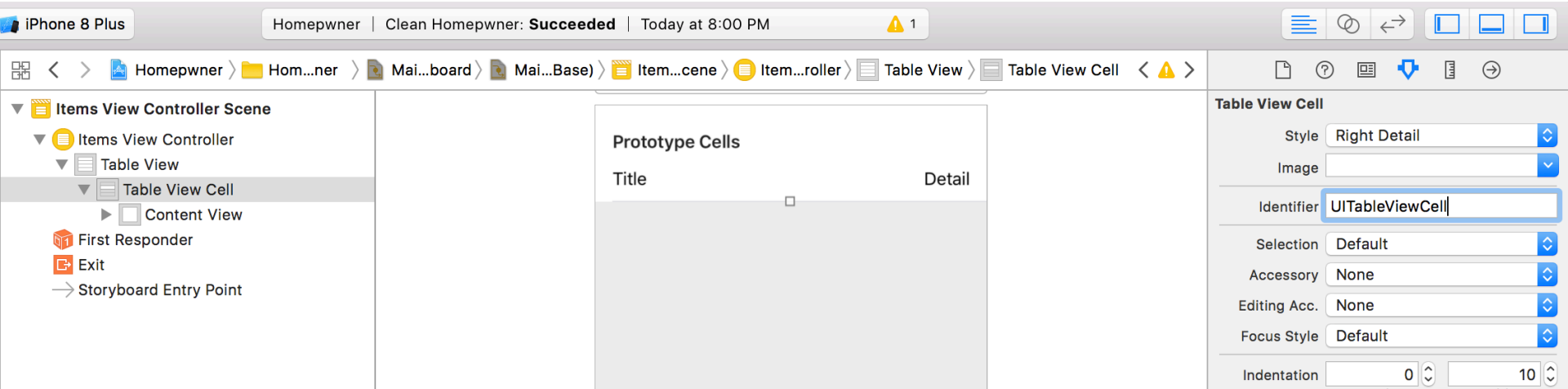
Reusing UITableViewCells

In Prototype Cells you can configure the different kinds of cells that you need for the associated table view.

Select the prototype cell and open its attributes inspector.

Change the Style to Right Detail (which corresponds to UITableViewCellStyle.value1)

and give it an Identifier of UITableViewCell.



Reusing UITableViewCells

Next, in ItemsViewController.swift, update tableView(_:cellForRowAt:) to reuse cells.

override func tableView(_ tableView: UITableView,

override func tableView(_ tableView: UITableView,
cellForRowAt indexPath: IndexPath) -> UITableViewCell {
// Create an instance of UITableViewCell, with default appearance
let cell = UITableViewCell(style: .value1, reuseIdentifier: "UITableViewCell")

// Get a new or recycled cell
“let cell = tableView.dequeueReusableCell(withIdentifier: "UITableViewCell",
for: indexPath)
...

The method dequeueReusableCell(withIdentifier:for:) will check the pool, or queue, of cells to see whether a cell with the correct reuse identifier already exists. If so, it will “dequeue” that cell. If there is not an existing cell, a new cell will be created and returned.

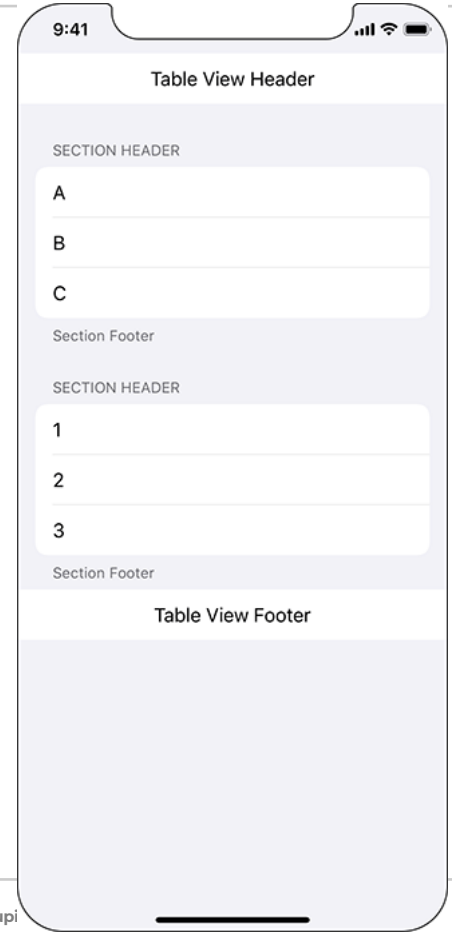
Build and run the application. The behavior of the application should remain the same. Reusing cells means that you only have to create a handful of cells, which puts fewer demands on memory.

Reusing UITableViewCells

Build and run the application.

Editing Table Views

“UITableView has an editing property, and when this property is set to true, the UITableView enters editing mode. Once the table view is in editing mode, the rows of the table can be manipulated by the user. Depending on how the table view is configured, the user can change the order of the rows, add rows, or remove rows



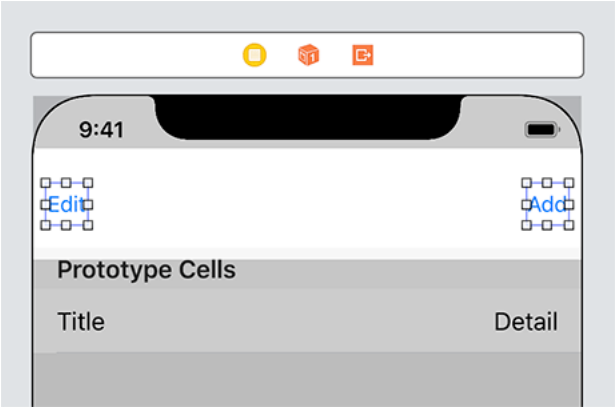
Reusing UITableViewCells

open Main.storyboard.

“From the library, drag a View to the very top of the table view, above the prototype cell. This will add the view as a header view for the table view. Resize the height of this view to be about 60 points. (You can use the size inspector if you want to make it exact.)

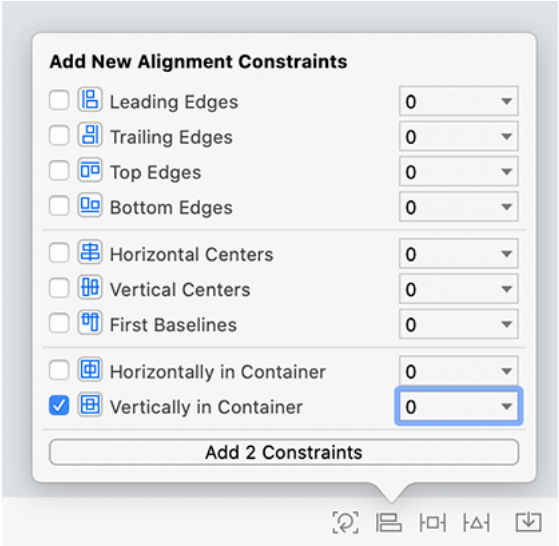
Now drag two Buttons from the library to the header view. Change their text and position them as shown:

“



Reusing UITableViewCells

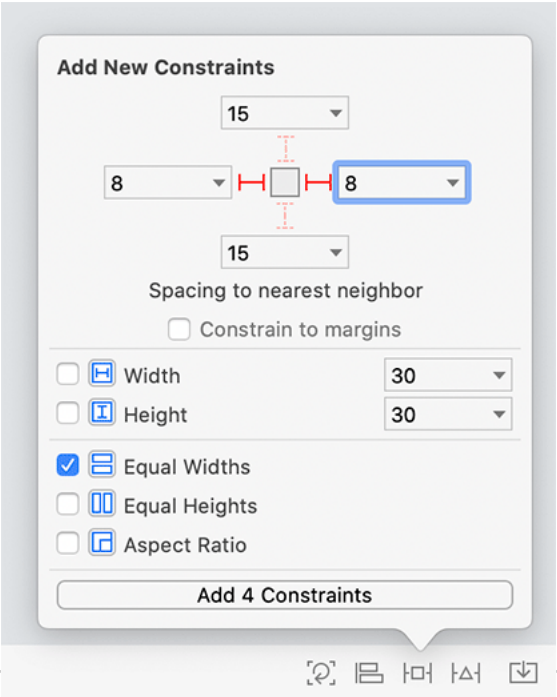
“Select both of the buttons and open the Auto Layout Align menu. Select Vertically in Container with a constant of 0, and then click Add 2 Constraints



Reusing UITableViewCells

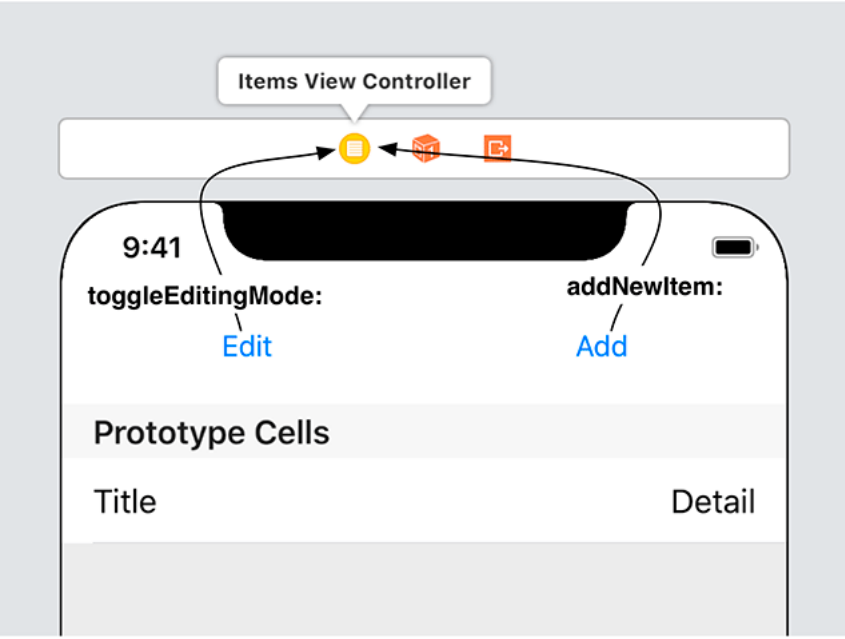
“Now open the Add New Constraints menu and configure it as shown”

“When you have done that, click Add 4 Constraints.”



Reusing UITableViewCells

"Finally, connect the actions for the two buttons as shown"



Adding Rows

“There are two common interfaces for adding rows to a table view at runtime.

A button above the cells of the table view: usually for adding a record for which there is a detail view. For example, in the Contacts app, you tap a button when you meet a new person and want to take down their information.

A cell with a green : usually for adding a new field to a record, such as when you want to add a birthday to a person’s record in the Contacts app. In editing mode, you tap the green next to add birthday.

Adding Rows / Deleting and Moving

Check ItemsViewController.swift,
ItemStore.swift

There are two common interfaces for adding rows to a table view at runtime.

A button above the cells of the table view: usually for adding a record for which there is a detail view. For example, in the Contacts app, you tap a button when you meet a new person and want to take down their information.

A cell with a green : usually for adding a new field to a record, such as when you want to add a birthday to a person's record in the Contacts app. In editing mode, you tap the green next to add birthday.

Design Patterns

“A design pattern solves a common software engineering problem. Design patterns are not actual snippets of code, but instead are abstract ideas or approaches that you can use in your applications. Good design patterns are valuable and powerful tools for any developer.

The consistent use of design patterns throughout the development process reduces the mental overhead in solving a problem so you can create complex applications more easily and rapidly. Here are some of the design patterns that you have already used:

Delegation: One object delegates certain responsibilities to another object. You used delegation with the UITextField to be informed when the contents of the text field change.

Data source: A data source is similar to a delegate, but instead of reacting to another object, a data source is responsible for providing data to another object when requested. You used the data source pattern with table views: Each table view has a data source that is responsible for, at a minimum, telling the table view how many rows to display and which cell it should display at each index path.

Model-View-Controller: Each object in your applications fulfills one of three roles. Model objects are the data. Views display the UI. Controllers provide the glue that ties the models and views together.

“Target-action pairs: One object calls a method on another object when a specific event occurs.